Logistics

Reading

- C Reference
- Bryant/O’Hallaron Ch 2.1-3

Goals

- Finish C overview
- Binary Representations / Notation
- Integers in binary
- Arithmetic operations

Project 1: Questions?

<table>
<thead>
<tr>
<th>Date</th>
<th>Event</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fri 2/7</td>
<td>C basics/Integers</td>
</tr>
<tr>
<td>Mon 2/10</td>
<td>Integers/Bits</td>
</tr>
<tr>
<td>Wed 2/12</td>
<td>Practice Exam</td>
</tr>
<tr>
<td></td>
<td>Lab04 Review</td>
</tr>
<tr>
<td></td>
<td>Project 1 Due</td>
</tr>
<tr>
<td>Fri 2/14</td>
<td>♡ Exam 1 ♡</td>
</tr>
</tbody>
</table>
Unsigned Integers: Decimal and Binary

- Unsigned integers are always positive:
  
  ```c
  unsigned int i = 12345;
  ```

- To understand binary, recall how decimal numbers “work”

**Decimal: Base 10 Example**  
Each digit adds on a power 10

- \(80,345 = 5 \times 10^0 + 4 \times 10^1 + 3 \times 10^2 + 0 \times 10^3 + 8 \times 10^4\)

**Binary: Base 2 Example**  
Each digit adds on a power 2

- \(11001_2 = 1 \times 2^4 + 0 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0\)

\[5 + 40 + 300 + 80,000 = 1 + 8 + 16 = 25\]

So, \(11001_2 = 25_{10}\)
Exercise: Convert Binary to Decimal

Base 2 Example:

\[ 11001 = 1 \times 2^4 + 1 \times 2^3 + 0 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 \]
\[ = 1 \times 16 + 1 \times 8 + 0 + 0 + 1 \]
\[ = 25 \]

So, \( 11001_2 = 25_{10} \)

Try With a Pal

Convert the following two numbers from base 2 (binary) to base 10 (decimal)

- 111
- 11010
- 01100001
Answers: Convert Binary to Decimal

\[ 111_2 = 1 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 \]
\[ = 1 \times 4 + 1 \times 2 + 1 \times 1 \]
\[ = 7_{10} \]

\[ 11010_2 = 1 \times 2^4 + 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 0 \times 2^0 \]
\[ = 1 \times 16 + 1 \times 8 + 0 \times 4 + 1 \times 2 + 0 \times 1 \]
\[ = 26_{10} \]

\[ 01100001_2 = 0 \times 2^7 + 1 \times 2^6 + 1 \times 2^5 + 0 \times 2^4 \]
\[ + 0 \times 2^3 + 0 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 \]
\[ = 0 \times 128 + 64 + 1 \times 32 + 0 \times 16 \]
\[ + 0 \times 8 + 0 \times 4 + 0 \times 2 + 1 \times 1 \]
\[ = 97_{10} \]

Note: last example ignores leading 0’s
The Other Direction: Base 10 to Base 2

Converting a number from base 10 to base 2 is easily done using repeated division by 2; keep track of **remainders**

**Convert 124 to base 2:**

\[
\begin{align*}
124 \div 2 &= 62 & \text{rem} & 0 \\
62 \div 2 &= 31 & \text{rem} & 0 \\
31 \div 2 &= 15 & \text{rem} & 1 \\
15 \div 2 &= 7 & \text{rem} & 1 \\
7 \div 2 &= 3 & \text{rem} & 1 \\
3 \div 2 &= 1 & \text{rem} & 1 \\
1 \div 2 &= 0 & \text{rem} & 1
\end{align*}
\]

▶ Last step got 0 so we’re done.
▶ Binary digits are in **remainders in reverse**
▶ Answer: 1111100
▶ Check:
\[
0 + 0 + 2^2 + 2^3 + 2^4 + 2^5 + 2^6 = 4 + 8 + 16 + 32 + 64 = 124
\]
Decimal, Hexadecimal, Octal, Binary

- Numbers exist independent of any writing system
- Can write the same number in a variety of bases
- C provides syntax for most common bases used in computing

<table>
<thead>
<tr>
<th>Base</th>
<th>Decimal</th>
<th>Binary</th>
<th>Hexadecimal</th>
<th>Octal</th>
</tr>
</thead>
<tbody>
<tr>
<td>Base 10</td>
<td>10</td>
<td>2</td>
<td>16</td>
<td>8</td>
</tr>
<tr>
<td>Mathematical</td>
<td>125</td>
<td>11111012</td>
<td>7D&lt;sub&gt;16&lt;/sub&gt;</td>
<td>175&lt;sub&gt;8&lt;/sub&gt;</td>
</tr>
<tr>
<td>C Prefix</td>
<td>None</td>
<td>0b...</td>
<td>0x..</td>
<td>0...</td>
</tr>
<tr>
<td>C Example</td>
<td>125</td>
<td>0b1111101</td>
<td>0x7D</td>
<td>0175</td>
</tr>
</tbody>
</table>

- **Hexadecimal** often used to express long-ish byte sequences
  - Larger than base 10 so for 10-15 uses letters A-F
- **Examine** `number_writing.c` and `table.c` for patterns
- **Expectation**: Gain familiarity with doing conversions between bases as it will be useful in practice
**Exercise: Conversion Tricks for Hex and Octal**

Examples shown in this week's HW, **What tricks are illustrated?**

<table>
<thead>
<tr>
<th>Decimal</th>
<th>Byte = 8bits</th>
<th>Byte by 4</th>
<th>Hexadecimal</th>
</tr>
</thead>
<tbody>
<tr>
<td>87</td>
<td>01010111</td>
<td>bin: 0101 0111</td>
<td>57 = 5*16 + 7</td>
</tr>
<tr>
<td></td>
<td></td>
<td>hex: 5 7</td>
<td>hex dec</td>
</tr>
<tr>
<td>60</td>
<td>00111100</td>
<td>bin: 0011 1100</td>
<td>3C = 3*16 + 12</td>
</tr>
<tr>
<td></td>
<td></td>
<td>hex: 3 C=12</td>
<td>hex dec</td>
</tr>
<tr>
<td>226</td>
<td>11100010</td>
<td>bin: 1110 0010</td>
<td>E2 = 14*16 + 2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>hex: E=14 2</td>
<td>hex dec</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Decimal</th>
<th>Byte = 8bits</th>
<th>Byte by 3</th>
<th>Octal</th>
</tr>
</thead>
<tbody>
<tr>
<td>87</td>
<td>01010111</td>
<td>bin: 01 010 111</td>
<td>127 = 1<em>8^2 + 2</em>8 + 7</td>
</tr>
<tr>
<td></td>
<td></td>
<td>oct: 1 2 7</td>
<td>oct dec</td>
</tr>
<tr>
<td>60</td>
<td>00111100</td>
<td>bin: 00 111 100</td>
<td>074 = 0<em>8^2 + 7</em>8 + 4</td>
</tr>
<tr>
<td></td>
<td></td>
<td>oct: 0 7 4</td>
<td>oct dec</td>
</tr>
<tr>
<td>226</td>
<td>11100010</td>
<td>bin: 11 100 010</td>
<td>342 = 3<em>8^2 + 4</em>8 + 2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>oct: 3 4 2</td>
<td>oct dec</td>
</tr>
</tbody>
</table>
Hexadecimal: Base 16

- Hex: compact way to write bit sequences
- One byte is 8 bits
- Each hex character represents 4 bits
- Each Byte is 2 hex digits

<table>
<thead>
<tr>
<th>Byte</th>
<th>Hex</th>
<th>Dec</th>
</tr>
</thead>
<tbody>
<tr>
<td>0101</td>
<td>57 = 5*16 + 7</td>
<td>87</td>
</tr>
<tr>
<td>0111</td>
<td>3C = 3*16 + 12</td>
<td>60</td>
</tr>
<tr>
<td>1100</td>
<td>E2 = 14*16 + 2</td>
<td>226</td>
</tr>
</tbody>
</table>

Hex to 4 bit equivalence

<table>
<thead>
<tr>
<th>Dec</th>
<th>Bits</th>
<th>Hex</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0000</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0001</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>0010</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>0011</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>0100</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>0101</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>0110</td>
<td>6</td>
</tr>
<tr>
<td>7</td>
<td>0111</td>
<td>7</td>
</tr>
<tr>
<td>8</td>
<td>1000</td>
<td>8</td>
</tr>
<tr>
<td>9</td>
<td>1001</td>
<td>9</td>
</tr>
<tr>
<td>10</td>
<td>1010</td>
<td>A</td>
</tr>
<tr>
<td>11</td>
<td>1011</td>
<td>B</td>
</tr>
<tr>
<td>12</td>
<td>1100</td>
<td>C</td>
</tr>
<tr>
<td>13</td>
<td>1101</td>
<td>D</td>
</tr>
<tr>
<td>14</td>
<td>1110</td>
<td>E</td>
</tr>
<tr>
<td>15</td>
<td>1111</td>
<td>F</td>
</tr>
</tbody>
</table>
Unix Permissions with Octal

- Octal arises associated with **Unix file permissions**
- Every file has 3 permissions for 3 entities
- Permissions are true/false so a single bit will suffice
- Octal historically used for this

- `ls -l`: long list files, shows permissions
- `chmod 665 somefile.txt`: change permissions of somefile.txt to those shown to the right
- `chmod 777 x.txt`: open to everyone
- **Symbolic chmod invocations are often preferred**
Character Coding Conventions

- Would be hard for people to share words if they interpreted bits as letters differently
- ASCII is an old standard for bit/character correspondence
- 7 bits per character, includes upper, lower case, punctuation

<table>
<thead>
<tr>
<th>Dec</th>
<th>Hex</th>
<th>Binary</th>
<th>Char</th>
<th>Dec</th>
<th>Hex</th>
<th>Binary</th>
<th>Char</th>
</tr>
</thead>
<tbody>
<tr>
<td>65</td>
<td>41</td>
<td>01000001</td>
<td>A</td>
<td>78</td>
<td>4E</td>
<td>01001110</td>
<td>N</td>
</tr>
<tr>
<td>66</td>
<td>42</td>
<td>01000010</td>
<td>B</td>
<td>79</td>
<td>4F</td>
<td>01001111</td>
<td>O</td>
</tr>
<tr>
<td>67</td>
<td>43</td>
<td>01000011</td>
<td>C</td>
<td>80</td>
<td>50</td>
<td>01010000</td>
<td>P</td>
</tr>
<tr>
<td>68</td>
<td>44</td>
<td>01000100</td>
<td>D</td>
<td>81</td>
<td>51</td>
<td>01010001</td>
<td>Q</td>
</tr>
<tr>
<td>69</td>
<td>45</td>
<td>01000101</td>
<td>E</td>
<td>82</td>
<td>52</td>
<td>01010010</td>
<td>R</td>
</tr>
<tr>
<td>70</td>
<td>46</td>
<td>01000110</td>
<td>F</td>
<td>83</td>
<td>53</td>
<td>01010011</td>
<td>S</td>
</tr>
<tr>
<td>71</td>
<td>47</td>
<td>01000111</td>
<td>G</td>
<td>84</td>
<td>54</td>
<td>01010100</td>
<td>T</td>
</tr>
<tr>
<td>72</td>
<td>48</td>
<td>01001000</td>
<td>H</td>
<td>85</td>
<td>55</td>
<td>01010101</td>
<td>U</td>
</tr>
<tr>
<td>73</td>
<td>49</td>
<td>01001001</td>
<td>I</td>
<td>86</td>
<td>56</td>
<td>01010110</td>
<td>V</td>
</tr>
<tr>
<td>74</td>
<td>4A</td>
<td>01001010</td>
<td>J</td>
<td>87</td>
<td>57</td>
<td>01010111</td>
<td>W</td>
</tr>
<tr>
<td>75</td>
<td>4B</td>
<td>01001011</td>
<td>K</td>
<td>88</td>
<td>58</td>
<td>01011000</td>
<td>X</td>
</tr>
<tr>
<td>76</td>
<td>4C</td>
<td>01001100</td>
<td>L</td>
<td>89</td>
<td>59</td>
<td>01011001</td>
<td>Y</td>
</tr>
<tr>
<td>77</td>
<td>4D</td>
<td>01001101</td>
<td>M</td>
<td>90</td>
<td>5A</td>
<td>01011010</td>
<td>Z</td>
</tr>
<tr>
<td>91</td>
<td>5B</td>
<td>10011101</td>
<td>[</td>
<td>97</td>
<td>61</td>
<td>10100001</td>
<td>a</td>
</tr>
<tr>
<td>92</td>
<td>5C</td>
<td>10011110</td>
<td>\</td>
<td>98</td>
<td>62</td>
<td>10100010</td>
<td>b</td>
</tr>
</tbody>
</table>
Unicode

World: why can’t I write 人 in my code/web address/email?

America: ASCII has 128 chars. Deal with it.

World: Seriously?

America: We invented computers. ’Merica!

World: ASCII Uses 7 bits per char, limited to 127 characters

UTF-8 uses 1-4 bytes to represent many more characters

Uses 8th bit in a byte to indicate extension to more than a single byte

Requires software to understand coding convention allowing broader language support

ASCII is a proper subset of UTF-8 making UTF-8 backwards compatible and increasingly popular

America: Deal with it. ’Merica!
Adding/subtracting in binary works the same as with decimal EXCEPT that carries occur on values of 2 rather than 10

**ADDITION #1**

\[
\begin{array}{c}
11 \quad \text{<-carries} \\
0100 1010 = 74 \\
+ 0101 1001 = 89 \\
\hline
1010 0011 = 163
\end{array}
\]

**SUBTRACTION #1**

\[
\begin{array}{c}
? \quad \text{<-carries} \\
0111 1001 = 121 \\
- 0001 0011 = 19 \\
\hline
VVVVVVVVVVVVVV
\end{array}
\]

**ADDITION #2**

\[
\begin{array}{c}
1111 \quad 1 \quad \text{<-carries} \\
0110 1101 = 109 \\
+ 0111 1001 = 121 \\
\hline
1110 0110 = 230
\end{array}
\]

**SUBTRACTION #2**

\[
\begin{array}{c}
12 \quad \text{x12 <-carries} \\
0111 0001 = 119 \\
- 0001 0011 = 19 \\
\hline
0110 0110 = 102
\end{array}
\]
Two’s Complement Integers: Representing Negative Values

- To represent negative integers, must choose a coding system.
- **Two’s complement** is the most common for this.
- Alternatives exist:
  - Signed magnitude: leading bit indicates pos (0) or neg (1).
  - One’s complement: invert bits to go between positive negative.
- Great advantage of two’s complement: arithmetic is identical to unsigned arithmetic.
- Hardware folks only need to make one set of units for both unsigned and signed arithmetic.
**Summary of Two’s Complement**

Short explanation: most significant bit is associated with a negative power of two.

<table>
<thead>
<tr>
<th>Unsigned Binary</th>
<th>Two's Complement (signed)</th>
</tr>
</thead>
<tbody>
<tr>
<td>7654 3210 : position</td>
<td>7654 3210 : position</td>
</tr>
<tr>
<td>ABCD EFGH : 8 bits</td>
<td>ABCD EFGH : 8-bits</td>
</tr>
<tr>
<td>A: 0/1 * +(2^7) <em>POS</em></td>
<td>A: 0/1 * -(2^7) <em>NEG</em></td>
</tr>
<tr>
<td>B: 0/1 * +(2^6)</td>
<td>B: 0/1 * +(2^6)</td>
</tr>
<tr>
<td>C: 0/1 * +(2^5)</td>
<td>C: 0/1 * +(2^5)</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>H: 0/1 * +(2^0)</td>
<td>H: 0/1 * +(2^0)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Unsigned Binary</th>
<th>Two's Complement (signed)</th>
</tr>
</thead>
<tbody>
<tr>
<td>7654 3210 : position</td>
<td>7654 3210 : position</td>
</tr>
<tr>
<td>1000 0000 = +128</td>
<td>1000 0000 = -128</td>
</tr>
<tr>
<td>1000 0001 = +129</td>
<td>1000 0001 = -127 = -128+1</td>
</tr>
<tr>
<td>1000 0011 = +131</td>
<td>1000 0011 = -125 = -128+1+2</td>
</tr>
<tr>
<td>1111 1111 = +255</td>
<td>1111 1111 = -1 = -128+1+2+4+..+64</td>
</tr>
<tr>
<td>0000 0000 = 0</td>
<td>0000 0000 = 0</td>
</tr>
<tr>
<td>0000 0001 = +1</td>
<td>0000 0001 = +1</td>
</tr>
<tr>
<td>0000 0101 = +5</td>
<td>0000 0101 = +5</td>
</tr>
<tr>
<td>0111 1111 = +127</td>
<td>0111 1111 = +127</td>
</tr>
</tbody>
</table>
Two’s Complement Notes

- Leading 1 indicates negative, 0 indicates positive
- All 0's = Zero
- Positive numbers are identical to unsigned

Conversion Trick
Positive -> Negative

- Invert bits, Add 1

Negative -> Positive

- Invert bits, Add 1

Same trick works both ways, implemented in hardware for the unary minus operator as in

```c
int y = -x;
```

```
~ 1001 1000 = negative, invert
-------------
0110 0111 = +103 inverted
+ 1
-------------
0110 1000 = +104 (original = -104)

~ 0110 1000 pos to neg
-------------
1001 0111 inverted
+ 1
-------------
1001 1000 = -104
original bits
```

Add Pos/Neg should give 0

```
1 1111 <-carries
0110 1000 = +104
+ 1001 1000 = -104
-------------
x 0000 0000 = zero
```
Overflow

- Sums that exceed the representation of the bits associated with the integral type **overflow**
- Excess significant bits are **dropped**
- Addition can result in a sum smaller than the summands, even for two positive numbers (!?)
- Integer arithmetic in fixed bits is a mathematical **ring**

Examples of Overflow in 8 bits

<table>
<thead>
<tr>
<th>ADDITION #3 OVERFLOW</th>
<th>ADDITION #4 OVERFLOW</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1111 111 &lt;-carries</td>
<td>1 0110 1010 = 362</td>
</tr>
<tr>
<td>1111 1111 = 255</td>
<td>x drop 9th bit</td>
</tr>
<tr>
<td>+ 0000 0001 = 1</td>
<td>0110 1010 = 106</td>
</tr>
<tr>
<td>----------------------</td>
<td>----------------------</td>
</tr>
<tr>
<td>1 0000 0000 = 256</td>
<td></td>
</tr>
<tr>
<td>x drop 9th bit</td>
<td></td>
</tr>
<tr>
<td>0000 0000 = 0</td>
<td></td>
</tr>
</tbody>
</table>
Underflow

- **Underflow** occurs in unsigned arithmetic when values go below 0 (no longer positive)
- Pretend that there is an extra significant bit to carry out subtraction
- Subtracting a positive integer from a positive integer may result in a **larger** positive integer (!?!)  
  - Integer arithmetic in fixed bits is a mathematical **ring**

**Examples of 8-bit Underflow**

```
SUBTRACTION #2 UNDERFLOW

?<-carries
0000 0000 = 0
- 0000 0001 = 1
----------
VVVVVVVVVVVVV
?<-carries
1 0000 0000 = 256 (pretend)
- 0000 0001 = 1
----------
VVVVVVVVVVVVV
x 2<-carries
0 1111 1110 = 256
- 0000 0001 = 1
----------
1111 1111 = 255
```
Overflow and Underflow In C Programs

- See `over_under_flow.c` for demonstrations in a C program.
- **No runtime errors** for under/overflow
- Good for hashing and cryptography
- Bad for most other applications: system critical operations should use checks for over-/under-flow
- See textbook Arianne Rocket crash which was due to overflow of an integer converting from a floating point value
- At assembly level, there are condition codes indicating that overflow has occurred
Endinaness: Byte ordering in Memory

- Single bytes like ASCII characters lay out sequentially in memory in increasing address
- Multi-byte entities like 4-byte ints require decisions on byte ordering
- We think of a 32-bit int like this
  
  Binary: 0000 0000 0000 0000 0001 1000 1110 1001  
  0 0 0 0 1 8 E 9

  Hex : 0018E9

  Decimal: 6377

- But need to assign memory addresses to each byte
  
  Little Endian: least significant byte early
  
  Big Endian: most significant byte early

- Example: Integer starts at address #1024

  Address
  
  LittleEnd: #1027 #1026 #1025 #1024
  Binary: 0000 0000 0000 0000 0001 1000 1110 1001
  0 0 0 0 1 8 E 9
  BigEnd: #1024 #1025 #1026 #1027
  Address
Little Endian vs. Big Endian

- Most modern machines use **little endian** by default
- Processor may actually support big endian
- Both Big and Little Endian have engineering trade-offs
- At one time debated hotly among hardware folks: *a la Gulliver’s Travels* conflicts
- Intel chips were little endian and “won” so set the basis for most modern use
- Big endian byte order shows up in **network programming**: sending bytes over the network is done in big endian ordering
- **Examine** `show_endianness.c` to see C code to print bytes in order
- Since most machines are little endian, will see bytes print in the revers order usually think of them
Output of *showendianness.c*

```
1 > cat showendianness.c
2 // Show endiannes layout of a binary number in memory Most machines
3 // are little endian so bytes will print leas significat earlier.
4 #include <stdio.h>
5
6 int main(){
7    int bin = 0b00000000000000000001100011101001; // 6377
8    // | | | | | | | |
9    // 0 0 0 0 0 1 8 e 9
10   printf("%d\n%x\n",bin,bin);               // show decimal/hex of binary
11   unsigned char *ptr = (unsigned char *) &bin; // pointer to beginning of bin
12   for(int i=0; i<4; i++){               // print bytes of bin from low
13       printf("%x ", ptr[i]);         // to high memory address
14   }
15   printf("\n");
16   return 0;
17 }
18 > gcc showendianness.c
19
20 > ./a.out
21 6377
22 18e9
23 e9 18 0 0

Notice: num prints with value 18e9 but bytes appear in reverse order e9 18 when looking at memory
```
Along with Addition and Subtraction, Multiplication and Division can also be done in binary.

Algorithms are the same as base 10 but more painful to do by hand.

This pain is reflected in hardware speed of these operations.

The Arithmetic and Logic Unit (ALU) does integer ops in the machine.

A clock ticks in the machine at some rate like 3Ghz (3 billion times per second).

Under ideal circumstances, typical ALU Op speeds are:

<table>
<thead>
<tr>
<th>Operation</th>
<th>Cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>Addition</td>
<td>1</td>
</tr>
<tr>
<td>Logical</td>
<td>1</td>
</tr>
<tr>
<td>Shifts</td>
<td>1</td>
</tr>
<tr>
<td>Subtraction</td>
<td>1</td>
</tr>
<tr>
<td>Multiplication</td>
<td>3</td>
</tr>
<tr>
<td>Division</td>
<td>&gt;30</td>
</tr>
</tbody>
</table>

Due to disparity, it is worth knowing about relation between multiply/divide and bitwise operations.

Compiler often uses such tricks: shift rather than multiply/divide.
Mangling bits puts hair on your chest

Below contrasts difference between logical and bitwise operations.

```c
int xl = 12 || 10; // truthy (Logical OR)
int xb = 12 | 10; // 14 (Bitwise OR)
int yl = 12 && 10; // truthy (Logical AND)
int yb = 12 & 10; // 8 (Bitwise AND)
int zb = 12 ^ 10; // 6 (Bitwise XOR)
int wl = !12; // falsey (Logical NOT)
int wb = ~12; // 3 (Bitwise NOT/INVERT)
```

▶ Bitwise ops evaluate on a per-bit level
▶ 32 bits for int, 4 bits shown

<table>
<thead>
<tr>
<th>Bitwise OR</th>
<th>Bitwise AND</th>
<th>Bitwise XOR</th>
<th>Bitwise NOT</th>
</tr>
</thead>
<tbody>
<tr>
<td>1100 = 12</td>
<td>1100 = 12</td>
<td>1100 = 12</td>
<td></td>
</tr>
<tr>
<td>1010 = 10</td>
<td>1010 = 10</td>
<td>^ 1010 = 10</td>
<td>~ 1100 = 12</td>
</tr>
<tr>
<td>-----------</td>
<td>-------------</td>
<td>-------------</td>
<td>-------------</td>
</tr>
<tr>
<td>1110 = 14</td>
<td>1000 = 8</td>
<td>0110 = 6</td>
<td>0011 = 3</td>
</tr>
</tbody>
</table>
Bitwise Shifts

- **Shift** operations move bits within a field of bits
- Shift operations are
  
  ```
  x = y << k; // left shift y by k bits, store in x
  x = y >> k; // right shift y by k bits, store in x
  ```

- All integral types can use shifts: long, int, short, char
- **Not applicable** to pointers or floating point
- Examples in 8 bits
  
  ```
  // 76543210
  char x = 0b00010111; // 23
  char y = x << 2; // left shift by 2
  // y = 0b01011100; // 92
  // x = 0b00010111; // not changed
  char z = x >> 3; // right shift by 3
  // z = 0b00000010; // 2
  // x = 0b00010111; // not changed
  char n = 0b10000000; // -128, signed
  char s = n >> 4; // right shift by 4
  // s = 0b11111000; // -8, sign extension
  // right shift >> is "arithmetic"
  ```
Shifty Arithmetic Tricks

- Shifts with add/subtract can be used instead of multiplication and division
- Turn on optimization: gcc -O3 code.c
- Compiler automatically does this if it thinks it will save cycles
- *Sometimes* programmers should do this but better to convince compiler to do it for you, comment if doing manually

### Multiplication

```
// 76543210
char x = 0b00001010; // 10
char x2 = x << 1; // 10*2
// x2 = 0b00010100; // 20
char x4 = x << 2; // 10*4
// x4 = 0b00101000; // 40
char x7 = (x << 3)-x; // 10*7
// x7 = (x * 8)-x; // 10*7
// x7 = 0b01000110; // 70
// 76543210
```

### Division

```
// 76543210
char y = 0b01101110; // 110
char y2 = y >> 1; // 110/2
// y2 = 0b00110111; // 55
char y4 = y >> 2; // 110/4
// y4 = 0b00011011; // 27
char z = 0b10101100; // -84
char z2 = z >> 2; // -84/4
// z2 = 0b11101011; // -21
// right shift sign extension
```
Showing Bits

- **printf() capabilities:**
  - `%d` as Decimal
  - `%x` as Hexadecimal
  - `%o` as Octal
  - `%c` as Character

- No specifier for binary
- Can construct such with bitwise operations
- Code pack contains two codes to do this
  - **printbits.c:** single args printed as 32 bits
  - **showbits.c:** multiple args printed in binary, hex, decimal

- Showing bits usually involves shifting and bitwise AND &
- Example from `showbits.c`

```c
#define INT_BITS 32

// print bits for x to screen
void showbits(int x){
    int mask = 0x1;
    for(int i=INT_BITS-1; i>=0; i--){
        int shifted_mask = mask << i;
        if(shifted_mask & x){
            printf("1");
        } else {
            printf("0");
        }
    }
}
```
Bit Masking

▶ Semi-common for functions to accept bit patterns which indicate true/false options
▶ Frequently makes use of bit masks which are constants associated with specific bits
▶ Example from earlier: Unix permissions might be...

```c
#define S_IRUSR 0b100000000 // User Read
#define S_IWUSR 0b010000000 // User Write
#define S_IXUSR 0b001000000 // User Execute
#define S_IRGRP 0b000100000 // Group Read

... 
#define S_IWOTH 0b000000010 // Others Write
#define S_IXOTH 0b000000001 // Others Execute
```

▶ Use them to create options to C functions like

```c
int permissions = S_IRUSR|S_IWUSR|S_RGRP;
chmod("/home/kauffman/solution.zip",permissions);
```